TP13 : Primalité

Objectifs du TP :

- Mettre en application des connaissances en informatique, arithmétique et probabilités.
- Apprendre à générer des objets aléatoires en C et en Ocaml.

Dans ce TP, on s'intéresse au problème de décision suivant :

$$\text{PREMIER} : \begin{cases} \textbf{Entrée} : \text{Un entier } n \in \mathbb{N}. \\ \textbf{Question} : n \text{ est-il premier ?} \end{cases}$$

L'objectif est de comprendre le fonctionnement des algorithmes étudiés en étant bien conscient que, puisqu'on travaille sur des entiers de petite taille (64 bits), tester la primalité de l'un d'entre eux de manière naïve et exacte se fait en temps très raisonnable. En pratique, on utiliserait plutôt ces algorithmes sur des entiers à plusieurs milliers de bits ce qui rend les méthodes naïves impraticables.

Partie 1 *Crible d'Eratosthène*

Les fonctions de cette partie seront implémentées en C. On y étudie des algorithmes déterministes permettant d'établir la primalité d'un entier n. Un algorithme naïf pour ce faire consiste à vérifier qu'aucun entier $d \leq \sqrt{n}$ autre que 1 ne divise n.

1. Ecrire une fonction `bool est_premier_naif(int64_t n)` déterminant si son entrée est un nombre premier selon cette stratégie.
2. Minorer la complexité pire cas de cette fonction. Qu'en penser ?

Dans les cas où on souhaite tester la primalité de k entiers tous inférieurs à une borne commune N, une façon d'amortir le temps de calcul pour chaque entier est de précalculer une bonne fois pour toutes tous les nombres premiers plus petits que N. A la place d'utiliser notre algorithme naïf pour tester la primalité de chaque entier de $[\![1, N]\!]$, on peut utiliser le crible d'Eratosthène.

3. Ecrire une fonction `bool* crible_eratosthene(int64_t N)` renvoyant un tableau à $N+1$ cases et contenant `true` en case i si et seulement si i est premier en utilisant le crible d'Erathostène.
4. Estimer la complexité temporelle et spatiale de ce crible. On admettra pour ce faire que

$$\sum_{\substack{p \leq N \\ p \in \mathbb{P}}} \frac{1}{p} \sim \log\log N$$

à l'infini. Une fois le tableau `crible_eratosthene(N)` calculé, quel est le coût de vérification qu'un entier inférieur à N est premier ?

Interlude *Syntaxe aléatoire*

En pratique, pour vérifier si un entier est premier, on procède souvent probabilistiquement. On donne dans cette partie quelques éléments de syntaxe C et Ocaml au sujet de la génération (pseudo-)aléatoire de nombres dans ces deux langages. Dans les deux cas, il faut commencer par initialiser le générateur de nombres pseudo-aléatoires (PRNG : pseudorandom number generator) à l'aide d'une graine. Il y a deux façons de faire cette initialisation, à choisir selon le comportement souhaité :

- Initialiser le générateur avec une graine connue fixe. Dans ce cas, à chaque exécution de votre code, les choix aléatoires effectués seront toujours les mêmes et votre algorithme probabiliste se comportera comme un algorithme déterministe. Ce comportement est utile lors des phases de debug.
- Initialiser le générateur avec une graine qui changera à chaque exécution. Dans ces conditions, votre algorithme probabiliste fera des choix différents à chaque exécution (c'est le comportement souhaité une fois le debug effectué). En Ocaml, l'appel à Random.self_init () permet d'obtenir ce comportement. En C, on fournit généralement au générateur pseudo aléatoire la date courante pour l'obtenir ; cette dernière se récupère à l'aide de time(NULL) et nécessite d'importer time.h.

Récapitulatif des commandes qui peuvent vous être utiles :

Action	Ocaml	C
Initialiser le générateur avec une graine fixe s	Random.init s	srand(s)
Initialiser le générateur avec une graine changeante	Random.self_init ()	srand(time(NULL))
Tirer un entier entre 0 inclus et n exclus	Random.int n	rand() % n
Tirer un booléen	Random.bool ()	rand() % 2 == 0
Tirer un flottant entre 0 et 1 inclus	Random.float 1.	(float)rand() / RAND_MAX

Pour retrouver les commandes adaptées en C, il suffit de se souvenir du fonctionnement de la fonction rand : cette fonction ne prend pas d'entrée et génère un entier aléatoirement entre 0 et une valeur maximale fixée dont le nom est RAND_MAX et dont la valeur dépend de la machine.

Partie 2 *Tests de primalité probabilistes*

Cette partie est à rendre pour le mardi 7/02 à 10h au format papier. Elle sera traitée au choix (entièrement) en C ou (entièrement) en Ocaml.

Le petit théorème de Fermat assure que, si n est un nombre premier alors pour tout a premier avec n (donc en particulier pour tout $a \in [\![1, n-1]\!]$) on a :

$$a^{n-1} \equiv 1 \bmod n$$

On dit qu'un entier n passe le test de Fermat pour un entier $a \in [\![1, n-1]\!]$ si $a^{n-1} \equiv 1 \bmod n$.

5. Redémontrer le petit théorème de Fermat.
6. Ecrire une fonction expo_modulaire prenant en entrée trois entiers naturels x, n, p et calculant efficacement $x^n \bmod p$. On justifiera que le nombre de produits modulaires effectués est en $O(\log n)$.
7. Les entiers non premiers qui passent le test de Fermat pour $a = 2$ s'appellent les nombres de Poulet. Ecrire en l'expliquant une fonction poulet prenant en entrée un entier n et affichant les nombres de Poulet inférieurs à n. Quels sont les nombres de Poulet inférieurs à 1000 ?
8. En exploitant le petit théorème de Fermat et en expliquant la démarche, concevoir un algorithme de type Monte Carlo à erreur unilatérale permettant de tester si un entier est premier. Ecrire une fonction est_premier_fermat implémentant cet algorithme qu'est le test de primalité de Fermat.

La réciproque du petit théorème de Fermat est fausse : il existe des entiers composés n tels que pour tout a premier avec n, $a^{n-1} \equiv 1 \bmod n$: on les appelle les nombres de Carmichael. Ainsi, le test de primalité de Fermat ne permet pas de distinguer les nombres premiers et les nombres de Carmichael.

9. Montrer que, si n n'est pas un nombre de Carmichael, alors la probabilité de faux positif sur n avec est_premier_fermat est inférieure ou égale à $1/2$. *Indication : Observer que* $\{a \in [\![1, n-1]\!] \mid a^{n-1} \equiv 1 \bmod n\}$ *est un sous groupe de* $(\mathbb{Z}/n\mathbb{Z})^\star$.

10. En ignorant l'influence des nombres de Carmichael, écrire une fonction est_presque_premier_fermat résolvant PREMIER et dont la probabilité de renvoyer oui alors que son entrée est composée est inférieure à 10^{-20}. Justifier la démarche.

Une variante du test de primalité de Fermat implémenté ci-dessus est le test de primalité de Miller-Rabin. Ce dernier repose sur le résultat R suivant :

Si n est un nombre premier impair tel que $n-1 = 2^s \times m$ avec m impair alors pour tout $a \in [\![1, n-1]\!]$ l'une des deux conditions suivantes est vérifiée :

$$(1) \quad a^m \equiv 1 \bmod n$$

$$(2) \quad \exists d \in [\![0, s-1]\!] \text{ tel que } a^{2^d \times m} \equiv -1 \bmod n$$

Un élément $a \in [\![1, n-1]\!]$ tel que ni (1) ni (2) ne soit vraie s'appelle un témoin de Miller-Rabin pour n.

11. On admet que si n est premier, alors les seules racines carrées de 1 modulo n sont 1 et -1. Montrer R.

On admet par la suite le théorème de Rabin : si n est un entier impair non premier, la probabilité qu'un entier aléatoire de $[\![1, n-1]\!]$ soit un témoin pour n est supérieure à $3/4$.

12. En exploitant ce fait, concevoir un algorithme Monte Carlo à erreur unilatérale déterminant si un entier n est premier avec une bonne probabilité. Justifier votre construction et indiquer la probabilité de faux positif.

13. Ecrire une fonction est_presque_premier_mr prenant en entrée un entier n et un entier k et indiquant si n est premier avec une probabilité d'erreur minorée par $(1/4)^k$. On explicitera son fonctionnement et on prendra garde à organiser les calculs de sorte à optimiser la complexité.

    Indication : Vous pouvez commencer par écrire une fonction qui détermine si a est un témoin de Miller-Rabin pour n étant donnés n, s, m et vérifier qu'elle vous indique bien que 2 et 3 sont témoins pour 561 mais pas 50 et que 5 est témoin pour 1373653 mais pas 2 ni 3.

Remarque : Les tests précédents permettent d'assurer efficacement qu'un entier n n'est pas premier. En revanche, cela ne fournit pas de factorisation de n. A l'heure actuelle, les meilleures algorithmes de factorisation restent assez inefficaces et c'est heureux car de la difficulté à factoriser des entiers découle la sécurité de certains systèmes de chiffrement comme RSA.